计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(7): 1816−1832, 2025

PetS: 针对参数高效 Transformer 模型的可扩展推理服务系统

魏学超[1,4] 周 哲[2] 徐盈辉[3] 张洁靖[4] 谢 源[5] 孙广宇[2]
[1]（北京大学信息科学技术学院 北京 100871）
[2]（北京大学集成电路学院 北京 100871）
[3]（复旦大学人工智能创新与产业研究院 上海 201203）
[4]（阿里巴巴达摩院 杭州 310023）
[5]（香港科技大学电子工程学院 香港 999077）
（xuechao.wei@pku.edu.cn）

PetS: A Scalable Inference Serving System for Parameter-Efficient Transformer

Wei Xuechao[1,4], Zhou Zhe[2], Xu Yinghui[3], Zhang Jiejing[4], Xie Yuan[5], and Sun Guangyu[2]
[1]（*School of Electronics Engineering & Computer Science*, *Peking University*, *Beijing* 100871）
[2]（*School of Integrated Circuits*, *Peking University*, *Beijing* 100871）
[3]（*Artificial Intelligence Innovation and Incubation Institute*, *Fudan University*, *Shanghai* 201203）
[4]（*Alibaba DAMO Academy*, *Hangzhou* 310023）
[5]（*Department of Electronic & Computer Engineering*, *The Hong Kong University of Science and Technology*, *Hong Kong* 999077）

Abstract Deploying Transformer models under the conventional pre-train-then-fine-tune paradigm is challenging for multi-task serving, because a full model copy for each downstream task must be maintained, quickly exhausting the storage budget. Recent algorithmic advances in parameter-efficient Transformer (PET) have shown enormous potential to mitigate the storage overhead. They share the pre-trained model among tasks and only fine-tune a small portion of task-specific parameters. Unfortunately, existing serving systems neither have flexible PET task management mechanisms nor can efficiently serve queries to different tasks in batches. Therefore, we propose PetS, a unified framework for multi-task PET serving. Specifically, different PET tasks are expressed by a unified representation in the same framework, which enables flexible PET task management. Based on the unified representation, we design a specialized PET inference engine to batch different tasks' queries together and execute them with task-agnostic shared operators and task-specific PET operators. Equipped with the PET inference engine, PetS is more scalable with respect to the number of tasks on a single GPU device. To further improve system throughput, we propose a coordinated batching strategy taking query length, PET task type as well as system load balancing together into consideration. To improve the throughput on multiple GPU instances, we also propose a PET-migration based load balancing strategy. We evaluate PetS on platforms with single GPU, including Edge/Desktop/Server GPUs. Comprehensive experiments demonstrate that PetS supports up to 26 times more concurrent tasks and improves the serving throughput by 1.53 times and 1.63 times on desktop and server GPU nodes, respectively. On multiple GPUs, our load-balancing strategy also provides up to 29% speedup.

Key words inference serving; parameter-efficient Transformer (PET); GPU; distributed system; machine-learning system

收稿日期: 2024-03-21; **修回日期**: 2025-02-18
基金项目: 国家自然科学基金项目(62032001); 高等学校学科创新引智计划项目 (B18001)
This work was supported by the National Natural Science Foundation of China (62032001) and the Higher Education Discipline Innovation Project (B18001).
通信作者: 孙广宇(gsun@pku.edu.cn)

摘　要　在多任务推理服务场景下使用基于预训练-微调范式的 Transformer 模型存在很多困难：服务端必须维护每个下游任务的完整模型副本，从而造成很大的存储和显存开销. 最近逐渐兴起的参数高效 Transformer (PET) 算法在不同的下游任务之间共享预训练模型，仅微调一小部分任务特定的模型参数，从而减少存储的开销. 然而，现有的后端服务系统既没有灵活的 PET 任务管理机制，也不能有效地跨任务进行输入的批量处理. 针对不同的下游任务，现有框架在多卡分布式场景下也难以提供良好的负载均衡机制. 因此，提出了 PetS，一个用于多任务 PET 推理服务的可扩展框架. 具体而言，不同的 PET 任务在算法上被抽象成一种统一表示形式. 基于这种统一表示，设计了一个专门的 PET 推理引擎，以批处理不同任务的输入，并使用任务无关的共享算子和任务特定的 PET 算子进行推理. 通过 PET 推理引擎，PetS 在单个 GPU 设备上可以支持更多的任务数量. 为了进一步提高系统吞吐量，提出了一种协同批处理（CB）策略，同时考虑了输入的长度、PET 任务类型以及系统负载平衡. 为了提升多卡部署的负载均衡，创新性地提出了基于 PET 实时迁移的负载均衡机制. PetS 在包括边缘端、桌面端和服务器端 GPU 等多个平台上都经过了评估. 全面的实验证明，PetS 支持多达 26 倍的并发任务，并将服务吞吐量在桌面和服务器 GPU 节点上分别提高了 1.53 倍和 1.63 倍. 在多 GPU 场景下，该负载均衡策略可以将吞吐量进一步提升 29% 之多.

关键词　推理服务；参数高效 Transformer；GPU；分布式系统；机器学习系统

中图法分类号　TP316

DOI: 10.7544/issn1000-1239.202440206　　**CSTR:** 32373.14.issn1000-1239.202440206

近年来，大规模预训练的 Transformer 模型已经彻底改变了人工智能领域. 受益于“预训练-微调”范式，诸如 BERT[1]、GPT 系列[2-4]、Roberta[5]、XLNet[6]、T5[7]、OPT[8] 等基于 Transformer 的模型架构在各种自然语言处理（NLP）任务上取得了领先的性能，包括问答、情感分类、文本分类和机器翻译等. 除了 NLP 任务，一些研究还将 Transformer 应用于计算机视觉任务[9-10]，与传统卷积神经网络相比，表现出同等甚至更优的性能. 研究表明，基于 Transformer 的大模型应用正在成为数据中心中的主流应用[11].

为了将 Transformer 模型应用于实际的场景，业界通常采用如图 1 所示的工作流程. 预训练模型的提供公司如 Google 等首先使用大规模数据集对 Transformer 模型（如 BERT[1] 和 GPT[2-4]）进行预训练（步骤①）. 无监督的预训练通常需要数天到数月的时间. 具备丰富的通用知识的预训练模型被提供给应用程序开发人员，他们会以有监督的方式在私有数据集上对预训练模型进行微调（步骤②），以适应特定下游任务的需求. 微调过程通常只需要几分钟到几小时即可完成. 经过微调的特定任务模型最终部署到云端或边缘服务器（步骤④）以处理输入查询.

然而，这样的工作流程在多任务服务场景中面临着可扩展性差的问题. 如果 GPU 显存无法容纳所有被调用的任务模型，传统的服务框架必须频繁地换入换出模型，导致服务吞吐量大大降低. 此外，由于输入查询需要调用不同的模型，我们无法对其进行批处理来提高服务吞吐量[12-14].

参数高效 Transformer（parameter-efficient Transformer, PET）算法部分地解决了这些问题. PET 算法在多个不同任务间共享预训练模型权重，并且仅对

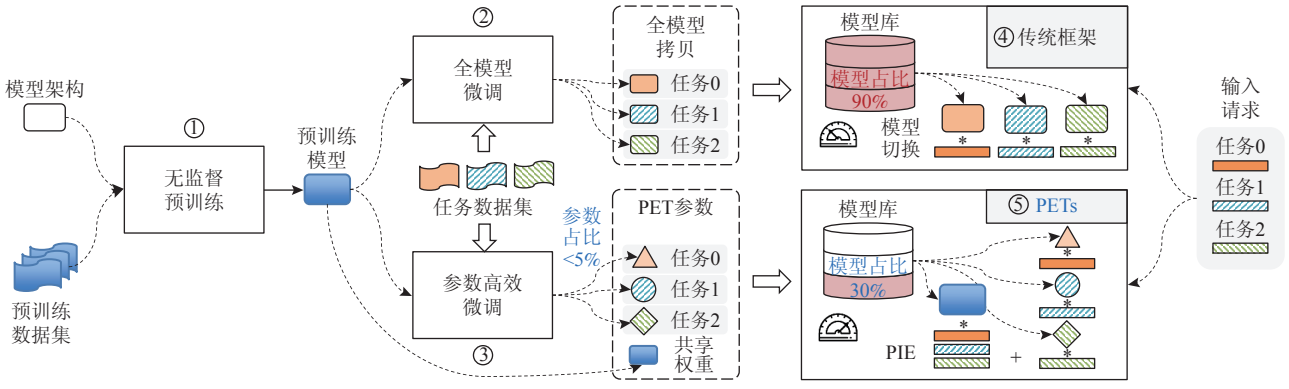


Fig. 1　Comparison of conventional workflow and PetS workflow

图 1　传统工作流和 PetS 工作流对比

每个下游任务微调一小部分特定于该任务的参数[15-19](步骤⑤). 这种方式可以大大减少存储开销，而模型的准确性仍然可与完整模型微调方法得到的准确性相媲美甚至更优.

然而，我们发现这些 PET 算法不能在现有的 Transformer 服务框架[20-21]上高效地运行. 一方面，由于缺乏 PET 任务管理机制和面向 PET 的推理引擎，框架不得不将 PET 参数合并到共享模型中，按照传统的推理方式进行全模型推理. 因此，GPU 内存占用没有减少. 另一方面，由于不同任务之间的权重差异和算法表示差异，无法对不同任务的查询进行批量处理，从而降低了系统的吞吐. 再者，当不同的下游任务被部署到多 GPU 节点时，由于不同任务被调用的频率不同，如何做多任务的负载均衡也是一个未被解决的问题.

为了充分发挥 PET 在多任务 Transformer 推理场景下的优势，本文提出了 PetS，这是一个具有出色可扩展性和性能的多任务 PET 服务统一框架. 本文进行了 5 个方面的创新：

1)为了解决 PET 算法的碎片化问题，本文提出了针对 PET 算法的统一表征，将任意 PET 模型推理过程分解为任务无关的共享操作和任务特定的 PET 操作.

2)基于 PET 算法的统一表征，本文提出了一个高效的 PET 任务管理机制，使用户能够灵活注册和加载调用不同的 PET 任务.

3)针对 PET 算法的计算特性，本文提出了一个高性能的 PET 推理引擎(PET inference engine, PIE)，能够将不同任务的查询进行批处理，并使用共享操作和轻量级 PET 操作分别推理执行，大大提高了服务吞吐量. PIE 支持单卡和多卡等多种推理场景.

4)为了进一步提高单实例的吞吐量，本文提出了一种协同批处理策略来调度任意输入查询，即具有不同序列长度和 PET 类型的查询.

5)为了提升多实例下的负载均衡性，本文提出了一种基于 PET 权重热迁移的动态负载均衡机制.

为了验证 PetS 系统的性能，本文在边缘/台式机/服务器 GPU 平台上全面评估了 PetS，并且涵盖了单卡和多卡的服务场景. 与传统的 Transformer 推理服务框架相比，PetS 支持多达 26 倍并发 Transformer 任务，并分别在台式机和服务器 GPU 上将服务吞吐量提高了 1.53 倍和 1.63 倍. 在多卡场景下，PetS 的负载均衡机制可以提升多达 29% 的吞吐. 因此，在多任务 Transformer 服务场景中，PetS 展现了降低服务部署成本和提高服务质量的巨大潜力.

1 背景介绍与研究动机

本节中主要介绍 Transformer 模型的背景知识以及多任务场景下 Transformer 模型推理服务的难点，并且讨论参数高效 Transformer 算法的原理和其在多任务 Transformer 推理场景下的挑战.

1.1 Transformer 网络架构

如图 2 所示，Transformer 模型通常由多个相同的 Transformer 块堆叠而成. 一个标准的 Transformer 块包括 3 个关键组件：多头注意力层、前馈神经网络和归一化层. 对于每个块，输入是一个由 n 个向量(词元)组成的序列，表示为 $\boldsymbol{X}\in\mathbb{R}^{n\times din}$，其中 n 和 din 分别是序列长度和输入特征维度. 3 个线性投影矩阵 $\boldsymbol{W}_{\boldsymbol{Q}}$，$\boldsymbol{W}_{\boldsymbol{K}},\boldsymbol{W}_{\boldsymbol{V}}\in\mathbb{R}^{din\times d}$ 将输入张量 $\boldsymbol{X}$ 投影到查询、键和值张量，表示为 $\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}\in\mathbb{R}^{n\times d}$(图 2 中的步骤 ①)，其中 d 表示特征向量维度. 将 $\boldsymbol{Q}$, $\boldsymbol{K}$, $\boldsymbol{V}$ 张量分割为多个“注意力头”(步骤 ②)，分别进行基于 Softmax 的自注意力计算(步骤 ③). 然后自注意力结果进行拼接(步骤 ④)并线性变换(步骤 ⑤)以生成多头注意力层结果. 经过跳层连接和层归一化之后，隐藏特征 $\boldsymbol{X}$ 被输入到前馈神经网络层，该层由 2 个全连接层计算(步骤 ⑥⑦). 第 1 层的输出应用了 GELU 激活函数.其中步骤①⑤⑥⑦为线性层.

Transformer 模型中，一个块的输出作为下一个块的输入. 在最后一个块之上，通常会有一个分类层(BERT 模型)或者语言模型头(GPT 模型)，为特定的下游任务生成最终结果.

1.2 多任务 Transformer 处理

在本文关注的基于 Transformer 模型的智能应用场景下，“任务”被定义为依赖特定的领域知识或者输出形式的输入请求的集合. 不同的下游任务往往要求模型具有不同的领域知识或者输出形式. 由于 Transformer 预训练模型权重是由通用语料进行训练得到的，其往往不具备应用开发者需要的特定领域知识. 如果将一个通用的模型应用于特定的场景，如智慧医疗[22]、智慧农业[23]、智慧金融[24]等，开发者通常需要使用任务相关的数据集对模型进行微调以提升其回答的准确性. 此外，即使 Transformer 模型具有一定的“少样本学习”能力，即通过在上下文中给出一些输入-输出示例来帮助模型理解任务的输出需求，但由于模型自身能力和上下文长度的限制，无法保证在各种复杂任务上提供较好的输出效果. 这时，通

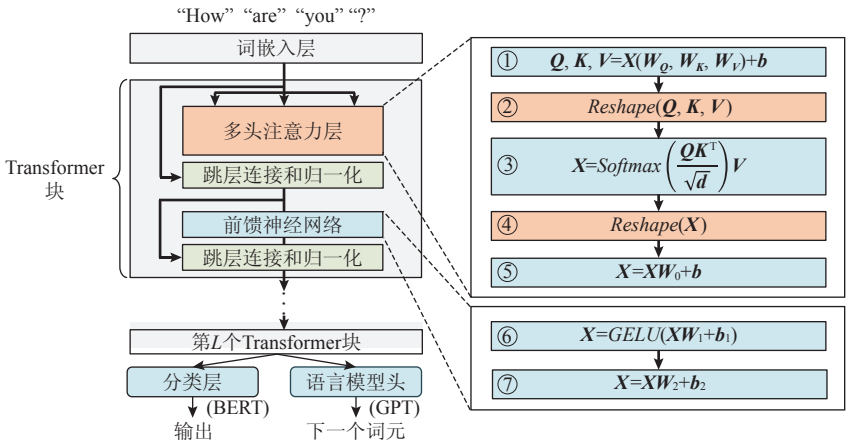


Fig. 2 Architecture of Transformer models

图 2 Transformer 模型架构

过对模型进行微调将会显著提升输出效果并且降低对上下文长度资源的消耗.

根据图 1 中步骤 ①②④所示的标准工作流程，每个下游任务都有自己的微调模型权重. 也就是说，存储/内存开销与部署的任务数量成正比. 在该图中，有 3 个任务部署到服务器上，占用了 3 倍的存储空间. 更重要的是，所有模型都应该缓存在 GPU 显存中，以快速响应不同的查询. 随着任务数量的增加，这种方法很容易超出 GPU 显存容量. 一个保守的方法是在调用某些任务时切换不同的模型，即从主机内存中通过 PCIe 加载新的模型权重. 然而，这种方法会降低系统的性能，因为模型切换开销相当大[25-26]. 此外，如果每个任务只有有限的输入查询，则由于批处理大小(batch size)不够大，无法高效利用计算资源.

尽管之前的模型服务系统/框架(如 INFaaS[25]，Nexus[26]，Rafiqi[27]，Triton[28]，TensorFlow Serving[29] 等)都强调了多任务 DNN 服务的能力，但实现多任务 Transformer 服务仍然具有挑战性. 原因主要有 2 个方面. 首先，Transformer 通常包含大量的参数以保证其充足的知识容量. 因此，存储和内存开销比传统的 DNN 要大得多，由此限制了可以服务的任务数量. 其次，现有的多任务推断框架/加速器假设并发 DNN 之间的计算/带宽需求不同. 因此，它们将计算密集型和内存密集型模型(或层)一起执行，以充分利用硬件资源. 然而，由于不同任务之间的 Transformer 块是同质的，通过共享不同模型的方式几乎无法提高系统吞吐量.

1.3 PET 网络

为了解决多任务 Transformer 推理时微调和存储效率问题，PET 被提出. PET 仅针对特定的下游任务微调一小部分参数而不是整个模型. 如图 1 所示，在参数高效的微调(步骤③)中，每个下游任务只需要存储少量 PET 的参数即可.

例如，4 种具有代表性的 PET，即 Adapters[16]，MaskBert[18]，Diff-Pruning[15]，BitFit[17]，每个任务只使用了额外的 0.5%～7.3% 的参数. 然而，它们仍然实现了与全模型微调相媲美甚至更高的准确性. 这些 PET 算法的介绍如下：

1)Adapters. Adapters 提出在预训练模型的某些层之间插入轻量级的可训练的模块，而预训练权重在任务之间共享. 形式上，假设预训练模型中的线性层使用输入特征 $\boldsymbol{X}_{\mathrm{t}}$ 以及预训练权重 $\boldsymbol{W}$ 和偏置 $\boldsymbol{b}$ 计算隐藏特征 $\boldsymbol{Y}_{\mathrm{t}}$ 即 $\boldsymbol{Y}_{\mathrm{t}}=\boldsymbol{X}_{\mathrm{t}}\boldsymbol{W}+\boldsymbol{b}$，那么一个典型的 Adapter 模块使用 2 个可学习的权重 $\boldsymbol{W}_{\mathrm{down}}\in\mathbb{R}^{d\times d_{\mathrm{m}}}$ 和 $\boldsymbol{W}_{\mathrm{up}}\in\mathbb{R}^{d_{\mathrm{m}}\times d}$ 操作隐藏特征，即 $\boldsymbol{Y}_{\mathrm{t}}=\boldsymbol{Y}_{\mathrm{t}}+\sigma(\boldsymbol{Y}_{\mathrm{t}}\boldsymbol{W}_{\mathrm{down}})\boldsymbol{W}_{\mathrm{up}}$，其中 σ 是激活函数. 由于瓶颈维度 $d_{\mathrm{m}}\ll d$，Adapter 模块相对于原模型的来说非常小. 每个任务仅需要约 7.3% 的新参数.

2)MaskBert. 基于 BERT 的彩票票证假设[30]，MaskBert 通过为每个权重矩阵学习二进制掩码来将预训练模型适应到下游任务中. 对于每个任务，预训练模型(包括分类层)被冻结，仅为每个权重矩阵学习包含约 5% 零元素的二进制掩码. 由于掩码是二进制的，MaskBert 仅增加了约 3% 的任务存储开销. 对于每个线性层，计算表示为 $\boldsymbol{Y}_{\mathrm{t}}=\boldsymbol{X}_{\mathrm{t}}(\boldsymbol{M}_{\mathrm{t}}\odot\boldsymbol{W})+\boldsymbol{b}$，其中 $\boldsymbol{M}_{\mathrm{t}}$ 表示任务特定的掩码.

3)Diff-Pruning. Diff-Pruning 也将预训练模型在任务之间共享，并且仅微调每个下游任务的一小部分“差异”. 每个任务仅增加约 0.5% 的新参数. 在推断过程中，这些差异参数($\boldsymbol{\delta}_{\mathrm{t}}$ 和 $\boldsymbol{b}_{\mathrm{t}}$)与预训练模型合并以构建一个任务特定的模型. 因此，主要计算变为 $\boldsymbol{Y}_{\mathrm{t}}=\boldsymbol{X}_{\mathrm{t}}(\boldsymbol{W}+\boldsymbol{\delta}_{\mathrm{t}})+(\boldsymbol{b}+\boldsymbol{b}_{\mathrm{t}})$.

4)BitFit. 除了任务特定的分类层外，BitFit 仅微

调线性层和归一化层的偏置项. BitFit 中的线性层计算为 $\boldsymbol{Y}_{\mathrm{t}}=\boldsymbol{X}_{\mathrm{t}}\boldsymbol{W}+\boldsymbol{b}_{\mathrm{t}}$, 其中 $\boldsymbol{b}_{\mathrm{t}}$ 是唯一的针对下游任务进行微调的参数.

除了上文提到的 4 种 PET 算法, 还有许多其他的 PET 算法变种如 LoRA[19] 等, 其工作流程至少与上述 4 种 PET 当中的一种类似. 文献 [31] 给出了目前所有 PET 算法的 3 种形式化的分类:

1)增量参数的方式. 通过引入额外的可训练神经模块或参数, 这些模块或参数在原始模型或过程中并不存在. 即原始模型参数集合为 $\theta=\{w_1,w_2,\cdots,w_N\}$, 通过引入新的参数即 $\Delta\theta=\{w_{N+1},w_{N+2},\cdots,w_M\}$来微调模型, 其中 $M\ll N$. Adapters 是这一类算法的代表之一.

2)基于指定参数更新的方式. 指定原始模型或过程中的某些参数为可训练, 而其他参数则被冻结. 即将可训练的参数集合定义为 W_{train}, $\Delta\theta=\{\Delta w_1,\Delta w_2,\cdots,\Delta w_N\}$, 当 $w_i\in W_{\mathrm{train}}$时, Δw_i表示参数 w_i微调更新后的差异值, 否则 $\Delta w_i=0$.Diff-Pruning 和 BitFit 为采用这种方式的 PET 算法.

3)重新参数化的方式. 通过变换将现有参数重新参数化为参数高效的形式. 将重新参数化的参数集合定义为 W_{rep}, 则将 W_{rep}集合内的参数 w_i替换为新的参数 $R(w_i)=\{u_1,u_2,\cdots,u_{N_i}\}$ 来得到新的模型参数. 即参数差异 $\Delta\theta=\left(\theta\backslash W_{\mathrm{rep}}\right)\cup U$, 其中集合 $U=\left\{u_j\middle|\exists w_i\in W, u_j\in R(w_i)\right\}$表示所有重新参数化后的参数集. MaskBert 可以被认为是 $R(w_i)=\{0\}$ 的一种重新参数化 PET 算法.

不失一般性, 本文以 4 种具体的 PET 算法, 即 Adapters, MaskBert, Diff-Pruning 和 BitFit 为例进行讨论. 这 4 种算法涵盖了上述 3 类 PET 算法形式. 因此, 其他的具体 PET 算法可以视作是本文讨论的 4 种算法的变种.

1.4 PET 网络的推理挑战

在为 T 个不同任务提供服务时, PET 算法将存储开销从原来的 $T\times\gamma$ 降低到 $T\times\eta+\gamma$, 其中 γ 和 η 分别表示完整模型参数和 PET 参数的数量. 由于 $\eta\ll\gamma$, 使用 PET 算法可以显著减少存储开销. 然而, 我们注意到 PET 算法的算法优势在传统 Transformer 服务框架中很难转化为实际的加速, 主要是由于以下挑战:

挑战 1. 框架无法灵活支持各种 PET 算法. 根据这些 PET 算法在论文中展示的结果, 不同的 PET 算法都有各自的优势任务, 4 种 PET 算法都不能作为通用选择. 也就是说, 应用开发人员倾向于为其下游任务选择最佳的 PET 算法[31-32]. 因此, 服务框架必须支持多种类型的 PET 算法. 然而, 当前的框架并没有针对不同的 PET 算法进行优化. 它们缺乏注册和灵活管理不同 PET 算法的机制.

挑战 2. GPU 内存占用并没有被减少. 为了使用传统推理框架来为 MaskBert 和 Diff-Pruning 等 PET 算法提供服务, 现有框架必须将特定任务的 PET 参数合并到共享模型中, 然后将新构建的模型加载到推理框架中进行服务. 因此, 并发任务仍然占用 $O(T\times\gamma)$ 的 GPU 内存, 限制了系统的可扩展性. 虽然可以在任务之间交换模型以解决 GPU 容量问题, 但由于模型交换开销, 这将导致吞吐量较低[25-26]. 因此, 理想的服务框架应该更灵活、高效地管理和计算 PET.

挑战 3. 众所周知, 批量推理是提高 DNN 服务系统吞吐量的关键. 然而, 由于 PET 参数和 PET 算法的差异, 传统的框架很难对不同任务的查询进行批处理(即使这些任务可能属于同一种 PET 算法), 以实现更高的吞吐量. 当每个并发任务只有少量查询需要处理时, 这个问题将更加突出.

挑战 4. 多任务场景下, 不同任务对应着不同的模型. 那么当模型被部署到不同 GPU 实例的时候, 不同 GPU 推理实例可能会因为模型被用户调用的频率不同出现负载不均衡的问题. 如何高效地实现负载均衡是一个亟须解决的难题.

2 PetS 框架

为了应对上述挑战, 我们提出了 PetS, 一个用于高效多任务 PET 服务的统一框架. 我们首先提出了一个统一的表示方法, 将所有常用的 PET 算法纳入一个框架中. 基于该统一表征, 我们开发了一个灵活的 PET 任务管理机制和一个专门的 PIE, 使得任务间和算法间的查询都可以进行批处理. 具体细节如下介绍.

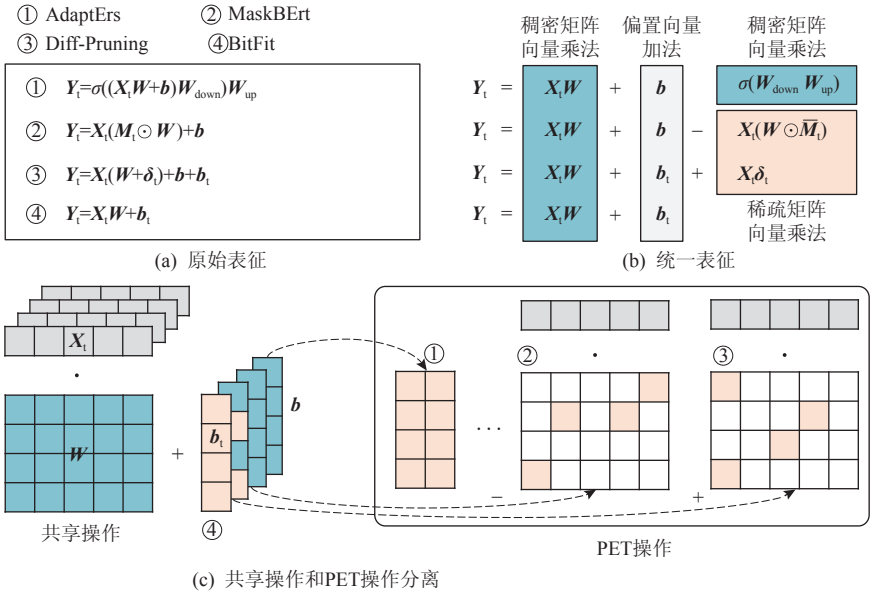
2.1 PET 算法的统一表示

由于 PET 算法具有不同的算法表示, 导致出现了“碎片化”问题. 为了解决这个问题, 我们提出了一个统一的表示方法, 即使用任务无关操作和任务相关操作来表达 PET 算法, 以帮助将它们纳入一个框架并实现批量推理.

如图 3 所示, 对于每个 PET, 我们将主要计算(如图 2 所示的步骤①⑤⑥⑦的线性层)分解为 3 个操作:

1)使用共享的预训练权重进行的稠密矩阵向量乘法操作.

2)使用共享的或任务相关的偏置向量进行的偏置向量加法操作.



(a) 原始表征 (b) 统一表征

(c) 共享操作和PET操作分离

Fig. 3 Unified representations of PET algorithm

图 3 PET 算法的统一表征

3)使用任务相关的 PET 参数进行的稀疏/稠密矩阵向量乘法操作.

由于所有的 PET 共享相同的预训练权重矩阵 $\boldsymbol{W}$, 第 1 个操作 $\boldsymbol{X}_t\boldsymbol{W}$可以进行批处理. 尽管涉及 PET 参数的计算无法在 PET 之间进行批处理, 但它只需要轻量级的计算, 开销比较小.

该算法的统一表示基于这样一个前提, 即根据 1.3 节对 PET 算法的 3 种分类定义得出, 所有 PET 算法微调后的模型参数为$\theta' = g(\theta, \Delta\theta)$, 其中 $\Delta\theta$为 PET 算法引入的模型参数, 函数g将修改融入原始参数θ的变换操作. 因此, 对于多任务 PET 推理, 即$\mathcal{Y} = f(\{\theta'_1, \theta'_2, \cdots, \theta'_n\}, \mathcal{X})$,其中函数$f$为 Transformer 的推理函数, $\{\theta'_1, \theta'_2, \cdots, \theta'_n\}$表示$n$个不同的模型, $\mathcal{X}$为输入请求集合, $\mathcal{Y}$为输出集合. 我们希望其可以转换为 $\mathcal{Y} = A \times f(\theta, \mathcal{X}) + B \times f\left(\{\Delta\theta'_1, \Delta\theta'_2, \cdots, \Delta\theta'_n\}, \mathcal{X}\right)$, 其中$A, B$为线性组合参数. 该转换的成立需要满足 2 个数学性质.

性质 1. PET 参数变换操作$g(\theta, \Delta\theta)$为线性组合函数, 即$g(\theta, \Delta\theta) = A \times \theta + B \times \Delta\theta$.

性质 2. 推理函数f对参数θ满足线性, 即$f(A \times \theta + B \times \Delta\theta, \mathcal{X}) = A \times f(\theta, \mathcal{X}) + B \times f(\Delta\theta, \mathcal{X})$.

其中性质 2 在整个模型的视角下难以达到. 但是由于 PET 算法仅仅改变模型的线性层(全连接层), 如果我们以逐层推理的视角来看, 性质 2 在全连接层的计算过程中是可以满足的. 对于性质 1, 采用增量参数式算法(如 Adapters, BitFit)和指定参数更新的 PET 算法(如 Diff-Pruning)是可以直接满足的. 并且线性参数$A = B = 1$, 对于重新参数式算法如 MaskBert, 可以通过引入额外的参数预处理, 将重新参数化变换转换为指定参数更新变换.

如图 3 所示, 对于 MaskBert, 我们使用等效的转换: $\boldsymbol{M}_t \odot \boldsymbol{W} = (1 - \bar{\boldsymbol{M}}_t) \odot \boldsymbol{W}$, 其中 $\bar{\boldsymbol{M}}_t$表示二进制掩码 $\boldsymbol{M}_t$ 的按位反转. 因此, 原始的矩阵向量乘法操作转换为 $\boldsymbol{X}_t\boldsymbol{W} - \boldsymbol{X}_t(\boldsymbol{W} \odot \bar{\boldsymbol{M}}_t)$. $\boldsymbol{W} \odot \bar{\boldsymbol{M}}_t$可以被视为类似于 Diff-Pruning 的稀疏权重差异. 由于 $\boldsymbol{\delta}_t$和 $\boldsymbol{W} \odot \bar{\boldsymbol{M}}_t$ 都是高稀疏度的稀疏矩阵(通常为 95% ~99.5% 的稀疏度), 这些 PET 操作可以使用稀疏算子高效地计算. 考虑到关于偏置项的向量加法操作只有很小的开销, 我们主要关注操作 1)和操作 3), 即稠密和稀疏的矩阵向量乘法操作.

统一的表示带来了 2 个主要的优势: 首先, 不论 PET 类型如何, 不同任务的查询可以在操作 1)中一起进行批处理. 其次, 这种统一的表示简化了 PET 任务的管理. 每个任务可以通过识别其共享模型标签、PET 类型和 PET 参数进行注册. 推理引擎可以以统一的方式加载这些 PET.

2.2 PET 推理框架设计

基于统一表示, 我们提出了 PetS 服务框架来支持 PET 任务的管理和服务. 图 4 展示了 PetS 框架的概览. PetS 具有 3 个主要组件: 任务管理器、参数仓库和 PET 推理流水线.PetS 的工作流程如下: 框架首先注册由开发者提交的 PET 任务(❶). 对于每个 PET 任务, 开发者需要提供预训练模型标签(如 bert-base-cased)、PET 参数(以压缩格式)和 PET 类型(例如 MaskBert). 任务管理器注册 PET 任务, 为每个提

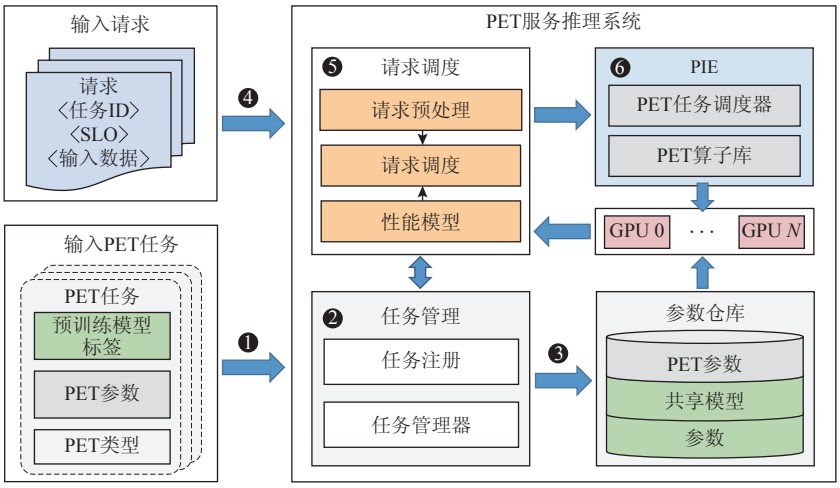


Fig. 4 Overview of PetS framework

图 4 PetS 框架概览

交的任务分配一个唯一的任务 ID(❷). PET 参数和预加载的共享模型参数都存储在参数仓库中(❸). 注册后, PIE 负责通过优化的 PET 推理流水线(❻)来处理经过请求调度(❺)的输入查询(❹).

2.2.1 PET 任务管理

PetS 的一个关键特性是灵活高效的 PET 任务管理机制. 这个机制主要由任务注册和任务加载模块提供.

1)任务注册. 任务注册模块根据用户提供的信息注册 PET 任务, 其中包括共享模型、任务特定参数和 PET 类型. 每个任务与其对应的预训练模型和支持的 PET 类型绑定, 形成一个三元组〈任务 ID, 共享模型标签, PET 类型〉, 其中任务 ID 是用于唯一标识每个 PET 任务的标识符. 三元组以任务 ID 作为键, 〈共享模型标签, PET 类型〉对作为值. 因此, 可以通过查询的任务 ID 索引每个任务的元数据.PET 任务一旦完成注册, 其 PET 参数将存储在参数仓库中.

2)任务加载. 在 PIE 调用 PET 任务之前, 任务加载模块首先加载共享模型参数; 否则, 任务加载器根据每个调用任务的任务 ID 在参数仓库中索引并获取 PET 参数. 考虑到 PET 参数的大小较小且共享模型只有 1 套, 所有的模型参数可以缓存到 GPU 显存中以便被快速调用.

2.2.2 PET 推理流水线

PetS 框架的核心是 PET 推理流水线, 它通过包括预处理、批处理调度和 PET 推理 3 个流水线步骤来处理查询.

1)预处理. 预处理模块分析输入的查询数据并对其进行格式化, 以便进行下一个查询批处理步骤. 首先, 预处理根据共享模型标签, 对输入数据进行分类. 接着提取每个查询的元数据, 如调用任务的 ID、序列长度、PET 类型等. 然后根据提取的元数据执行一些初步的数据预处理操作, 例如对相同 PET 任务的查询进行分组. 最后, 将预处理后的输入查询与提取的元数据一起根据目标共享模型分派到不同的队列中, 以进行进一步的调度.

2)批处理调度. 批处理是提高系统吞吐量的有效方法. 尽管 PetS 基于算法的统一表示来支持跨任务和跨算法的批处理, 但是由于 PET 类型和序列长度上存在异构性, 仍然会影响批处理的效果. 批处理调度器模块用于克服输入的异构性带来的挑战.

2.2.3 PIE

1)PIE 工作流程. 批处理查询最终被传入 PIE. 图 5 展示了 PIE 的基本工作流程. 在图中, 我们使用不同的纹路表示批次中查询的 PET 类型. 例如, 任务 0 属于 MaskBert, 而任务 2, 4 属于 Diff-Pruning. PIE 按以下方式开始每个 Transformer 层的计算: PIE 使用所有输入张量和共享权重 $\boldsymbol{W}$ 执行 GEMM 批处理计算(❶). PIE 通过在查找表中搜索其 *task_id*, 获取每个查询的 *PET_type* 属性. 批处理后的输入也根据任务 ID 被切割为几个小批次(任务内组批)(❷). 然后, PIE 根据获取的 PET 类型得到 PET 算子(❸)和 PET 参数(❹). PIE 按顺序在每个切片的小批次上执行 PET 运算. 这些 PET 算子还负责将 PET 结果合并到共享输出中(❺). 需要注意的是, PIE 还需要执行其他操作(如自注意力算子等), 但为了陈述方便, 这些操作在图中未显示.

2)PET 运算符库. 根据图 3 中的统一表示, PET 任务依赖于不同的 PET 操作. MaskBert 和 Diff-Pruning 涉及稀疏矩阵乘法, 因为它们的 PET 参数具有高度稀疏性. Adapter 执行轻量级的密集矩阵乘法算子,

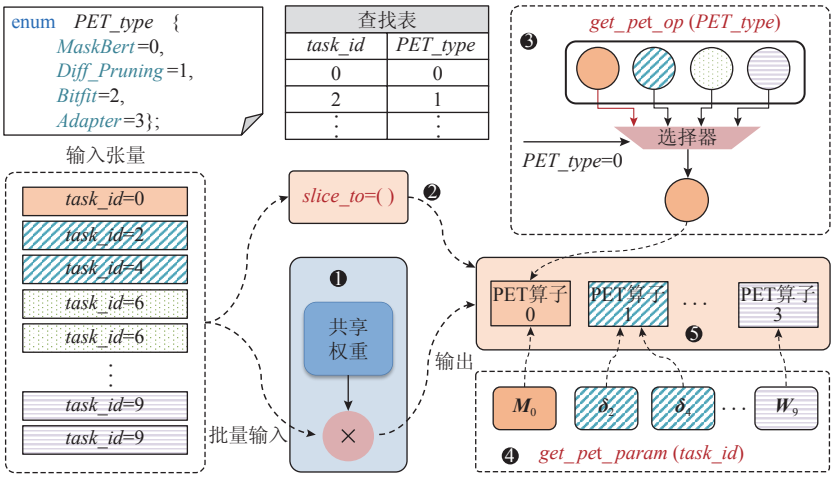


Fig. 5 Overview of PET inference engine

图 5 PET 推理引擎概览

BitFit 只需要一个向量加法操作. 因此, PIE 提供了一个算子库, 其中包含高性能的稠密和稀疏算子的实现. 如果其他新兴的 PET 算法可以适用于统一表示, 开发者也可以为其实现新的 PET 操作.

3) PET 任务调度器. PET 任务调度涉及 2 个方面. 一方面是针对每个 GPU 实例的. 在每个层的推理过程中, 不同任务的 PET 操作之间没有数据依赖关系, 因此可以并行运行. 给定系统允许的并行性(例如 GPU 上的 CUDA 流的数量), PET 任务调度器尽可能地利用并行性来调度 PET 操作. 另一方面是针对多 GPU 实例场景的. PET 任务调度器通过 PET 参数动态热迁移的方式在多个 GPU 实例之间实现负载均衡. 这些调度策略将在后文中进行详细介绍.

2.3 系统优化策略

2.3.1 协同批处理调度

在实际场景中, 针对文本输入进行处理的 Transformer 模型(对于采用图像[10]等作为输入的 Transformer 模型, 本文不作探讨), 其输入查询通常具有可变的序列长度(如 5~500[12]). 如果我们将短的查询输入和长的查询输入进行批处理, 那么短的查询就必须进行零填充(zero padding), 导致了无用的计算. 现有框架如 TurboTransformers[12]已经在 BERT 任务上提出这个问题的优化解决方案. 然而, 对于 PetS, 我们不仅需要考虑共享操作, 还要考虑 PET 操作. 因此, 我们提出了一种协同批处理(coordinated batching)策略, 在批处理过程中协调这 2 个部分.

1) 问题定义. 假设有 R 个查询, 即 $Q=\{x_0,x_1,\cdots,x_{R-1}\}$, 涉及 T 个不同的任务. 我们将查询分成 M 个批次, 对于每个批次, 我们用 $\alpha[N][L]$ 表示批处理 N 个查询的共享模型延迟, 其中最大长度为 L. 同时, 一个 PET 操作符需要 $\beta[pt][n][l]$ 延迟来处理 n 个查询的 PET 项, 其中 pt 用来索引不同的 PET_type, l 是这 n 个查询的最大长度. 执行延迟可以估计为

$$Batch_{Latency(B_i)}=\alpha[N_i][L_i]+\sum_{j=0}^{t_i-1}\beta[pt_{ij}][n_{ij}][l_{ij}], \tag{1}$$

其中, B_i 表示第 i 个批次, 并假设批次 i 中有 t_i 个不同的任务. 对于批次 i 中的第 j 个任务, 有 n_{ij} 个查询形成一个小批次, 由索引为 pt_{ij} 的 PET 操作符处理, 这 n_{ij} 个查询的最长序列长度为 l_{ij}. 由于所有的 M 个批次, 即 $\mathcal{B}=\{B_0,B_1,\cdots,B_{M-1}\}$ 是连续执行的, 我们可以进一步估计总延迟

$$Total_{Latency(\mathcal{B})}=\sum_{i=0}^{M-1}Batch_Latency(B_i)=\sum_{i=0}^{M-1}\alpha[N_i][L_i]+\sum_{i=0}^{M-1}\sum_{j=0}^{t_i-1}\beta[pt_{ij}][n_{ij}][l_{ij}]. \tag{2}$$

2) 总延迟由共享操作和 PET 操作共同决定. 为了协调这 2 个部分, 我们提出了一个 2 步的协同批处理策略. 如图 6 所示, 在第 1 步中, 我们使用 PET 算子的性能模型, 即 β 模型为每个任务生成“任务内批量”(*mini-batch*). 在第 2 步中, 通过将这些任务内批量在任务之间进行组合, 生成“任务间批量”(*macro-batch*), 这一步主要用到共享算子的性能模型, 即 α 模型. 在 2 个步骤中, 我们对输入查询按长度进行排序, 并使用动态规划来找到最佳切分位置.

首先, 如算法 1 所示, 调度算法将具有相同任务 ID 的查询进行聚类并按序列长度进行排序. 在算法 1 中, 我们使用 $state[i]$ 来记录在批处理第 i 个查询时 PET 操作的最小延迟. 我们使用 $split_idx_list$ 来记录

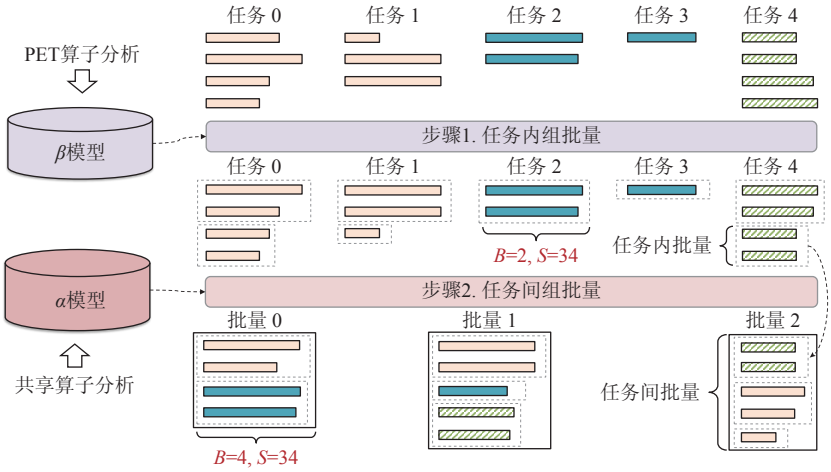


Fig. 6 Coordinated batching strategy

图 6 协同批处理策略

切分位置. 以下公式展示了 Bellman 方程:

$$state[i]=\min_{0<j\leqslant i}(state[j-1]+\beta[pt_{ij}][i-j+1][l_{ij}]). \quad (3)$$

根据这个 Bellman 方程使用动态规划算法, 将每个任务的查询分成多个 *mini-batch*.

算法 1. 组内批处理策略.

输入: 任务数量 T, 队列 $Q=\{X_0,X_1,\cdots,X_T\}$, PET 算子延迟模型(β模型);

输出: 组合后的多个 *mini-batch*.

① for $i\leftarrow 0$ until T

② 建立动态规划状态向量 $state[n_i+1]$, $state[0]=0$;

③ 根据 X_i 中请求的长度进行升序排序;

④ 建立 $split_idx_list[n_i+1]$, $pt=get_pet_type(i)$;

⑤ for $j\leftarrow 1$ to n_i, $min_cost=INF$

⑥ for $k\leftarrow 1$ to j

⑦ $tmp=state[k-1]+\beta[pt][j-k+1][X_i[j].len]$;

⑧ if $tmp<min_cost$

⑨ $min_cost=tmp.split_idx=k-1$;

⑩ end if

⑪ end for

⑫ end for

⑬ $state[j]=min_cost$, $split_idx_list[j]=split_idx$;

⑭ end for

⑮ 根据 *split_idx_list* 将所有输入请求切分到多个 *mini-batch*;

⑯ 返回多个 *mini-batch*.

再如算法 2 所示, Bellman 方程只考虑共享算子, 其延迟由 α 模型估计. 算法 2 不是调度单个查询, 而是调度 *mini-batches*:

$$state[i]=\min_{0<j\leqslant i}(state[j-1]+\alpha[batch_size][L_j]),$$

其中 $state[i]$ 记录批处理第 1 个 i 个 *mini-batch* 的最小延迟. L_j 表示第 j 个 *mini-batch* 的最大序列长度. *batch_size* 表示从 *mini-batch i* 到 *mini-batch j* 的总查询数. 通过动态规划的作用, 将多个 *mini-batch* 分配给多个 *macro-batch*.

算法 2. 组间批处理策略.

输入: 多个 *mini-batch*, 共享算子性能模型(α模型);

输出: 组合后的多个 *macro-batch*.

① 根据 batch 内最长的序列长度对多个 *mini-batch* 排序;

② 建立状态向量 $state[|MiB|+1]$, $state[0]=0$;

③ 建立 $sum[|MiB|+1]$, $sum[i]$ 记录了前 i 个 *mini-batches* 的总查询数量;

④ for $i\leftarrow 1$ to $|mini\text{-}batch|$, $min_cost=INF$

⑤ for $j\leftarrow 1$ to i

⑥ $bs=sum[i]-sum[j-1]$;

⑦ $tmp=state[j-1]+\alpha([bs][MiB[i].max_seq_len])$;

⑧ if $(tmp<min_cost)$

⑨ $min_cost=tmp, split_idx=j-1$;

⑩ end if

⑪ end for

⑫ $state[i]=min_cost, split_idx_list=split_idx$;

⑬ end for

⑭ 根据 *split_idx_list* 将输入的多个 *mini-batch* 切分到多个 *macro-batch*;

⑮ 返回多个 *macro-batch*.

2.3.2 多实例负载均衡调度

在多 GPU 场景下，不同 GPU 实例会加载共享的模型权重以及不同的 PET 任务. 然而，由于不同下游任务在一段时间内会存在被调用频次不同的问题，引起不同 GPU 实例间的负载不均衡. 一些实例可能会过载而影响服务质量，另一些实例可能会面临空转的问题造成资源的浪费. 为了缓解这些问题，本文结合 PET 算法的特性，提出了基于 PET 热迁移的负载均衡策略.

如图 7 所示，对于不同的 GPU 实例，都有一个输入队列来记录不同任务的负载情况. 注意输入队列是一个虚拟的概念，用于记录一段时间窗口内的所有查询输入，而不是在某个时间节点下该 GPU 实例的输入情况. 高负载 GPU 实例中将一些任务迁移到低负载实例中来提升均衡性. 由于 PET 参数量非常少，这样的迁移是高效的.

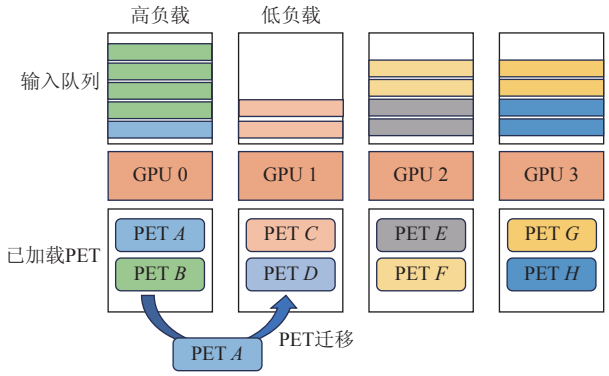


Fig. 7 Load-balancing based on PET heat migration

图 7 基于 PET 热迁移的负载均衡

如算法 3 所示，本文提出一个贪心的算法来求解如何调度 PET 的分布使系统达到一个尽可能负载均衡的状态. 算法分多轮迭代，在每轮迭代中，算法统计每个 GPU 实例中任务被调用的总次数(行③)，然后进行排序(行④). 接着尝试将最高负载的 GPU 实例中的任务迁移到最低负载的 GPU 实例中(行⑥)，并且检查这个操作是否使得调度后的最大调用次数差距减少(行⑧). 重复这个过程直到无法继续减少这个差值为止，并且输出最终的任务映射关系. 根据这个映射关系，PetS 会启动 PET 迁移程序进行最终的 PET 权重的迁移. 这个负载均衡的过程可以设定固定的频率(比如每分钟执行 1 次)来定期执行，以减少迁移对总体性能的负面影响.

算法 3. 基于 PET 动态迁移的负载均衡策略.

输入：GPU 实例数 N，所有 GPU 实例中的 PET 任务映射集合 $M=\{T_0,T_1,\cdots,T_{N-1}\}$，其中 $T_i=\{t_0^i,t_1^i,\cdots,t_{n_i-1}^i\}$ 表示第 i 个 GPU 实例中 n_i 个任务在一定时间窗口内被调用的次数；

输出：负载均衡后的任务映射集合.

① $\delta=INF$; /*实例中任务调用次数最大差值*/

② while true

③ 统计每个 GPU 实例中总的任务调用次数 $X=\{X_0,X_1,\cdots,X_{N-1}\}$, $X_i=\sum_{j=0}^{n_i-1}t_j^i$;

④ 对 X 进行升序排序得到 $\{X'_0,X'_1,\cdots,X'_{N-1}\}$;

⑤ $\delta'=X'_{N-1}-X'_0$;

⑥ 取出 X'_{N-1} 对应的任务队列中调用次数最低的任务放到 X'_0 对应的任务队列中;

⑦ 重新统计每个 GPU 实例中的总任务调用次数，排序得到 $\{X''_0,X''_1,\cdots,X''_{N-1}\}$;

⑧ $\delta''=X''_{N-1}-X''_0$; /*计算调度后的最大任务调用次数差别*/

⑨ if $\delta''\geqslant\delta'$ /*无法调度到更优，调度终止*/

⑩ 返回上一次调度结果;

⑪ else

⑫ continue; /*继续调度*/

⑬ end if

⑭ end while

另一个值得注意的问题是，算法 3 是根据一定的间隔来进行多轮的负载均衡调度. 对于固定输入分布的场景，通过设置固定间隔的方式就可以达到比较理想的效果. 但是对于输入请求的分布存在较大变化的场景，固定的迁移间隔可能并不能达到最优的效果. 为了应对这个问题，可以考虑引入一个简单的策略实现动态迁移频率的调整.

对于不同的 GPU 实例，我们首先定义其平均负载 L，即一段时间内每个 GPU 实例的任务队列的平均长度. 我们将系统的负载不均衡程度 ϕ 定义为最低平均负载和最高平均负载的比值即 $\phi=\frac{L_{\min}}{L_{\max}}$. 当 ϕ 值越接近 1 时说明负载均衡较好，越接近 0 时说明负载均衡越差. 在每次迁移前对比 ϕ 值与允许的最差不均衡阈值 θ，若 $\phi<\theta$ 则需要增加迁移频率来提升负载均衡，即减少迁移间隔. 新的迁移间隔 $\phi'=\max\left(\phi_{\min},\frac{\phi}{2}\right)$，其中 $\phi_{\min}$ 为迁移间隔的最小值. 若 $\phi\geqslant\theta$ 则说明目前的负载均衡情况较好，可以适当增加迁移间隔来降低迁移开销. 即 $\phi'=\max(\phi+\Delta\phi,\phi_{\max})$，其中 $\Delta\phi$ 为每次调整间隔的增量，$\phi_{\max}$ 为最大允许的迁移间隔.

2.4 PetS 框架实现

PetS 框架分为前端和后端，我们使用 Python 实

现 PetS 的前端,用于描述共享模型任务管理,并使用 C++实现后端查询调度和推理服务.

理论上 PetS 可以与 HuggingFace Transformers 库兼容的推理框架作为后端引擎进行集成,例如 TurboTransformers[12], LightSeq[21] 等. 这些框架需要进行修改以支持 PET 算子和 PET 任务调度器. 本文中, PetS 使用基于 TurboTransformers 实现的后端推理引擎 .

PetS 利用高性能的稀疏矩阵乘法实现稀疏 PET 算子[32].

3 实验验证

3.1 实验设置

1)共享模型. 为了验证 PetS 的性能,我们选择 Bert-base, Bert-large, DistillBERT 这 3 种 Transformer 模型来进行测试,其配置如表 1 所示.

Table 1 Configuration Parameters of Shared Model

表 1 共享模型的配置参数

模型	层数	Head 数	隐层长度	中间层长度	参数总量
DistillBert	6	12	768	3 072	66×10^6
Bert-base	12	12	768	3 072	110×10^6
Bert-large	24	16	1 024	4 096	340×10^6

2)PET 任务. PetS 目前支持 4 种不同的 PET 算法,用于测试 PET 算法的配置,如表 2 所示. Adapters 算法的 Bottleneck 层维度设置为 64, MaskBert 和 Diff-Pruning 的稀疏度分别设置为 95% 和 99.5%. 这些参数设置符合原论文中的默认配置.

Table 2 PET Configuration

表 2 PET 配置

PET 类型	配置	主要 PET 参数
Adapter	*Bottleneck* = 64	$\boldsymbol{W}_{\text{down}}$, $\boldsymbol{W}_{\text{up}}$
MaskBert	95% 稀疏	二值化 mask
Diff-Pruning	99.5% 稀疏	稀疏的 PET 权重矩阵
BitFit		线性层/归一化层和分类层的偏置项

3)硬件平台. 我们在边缘/桌面/服务器平台上评估 PetS, 即 Jetson-TX2(8 GB 内存,由 CPU 和 GPU 共享)、GTX-1080Ti-11GB(Intel Xeon E5-2690 CPU)和 Tesla-V100-32GB (Intel Xeon Golden 5220 CPU, 双 CPU). V100 平台安装了 CUDA-10.1, 1080Ti 平台安装了 CUDA-11.3.TX2 平台刷写了 Jetpack 4.4.1, 其中包含 CUDA-10.2.

3.2 实验结果

3.2.1 支持的最大任务数

首先,我们通过比较能够支持的最大任务数来展示 PetS 的出色可扩展性,并且与传统的顺序服务(sequential serving, SeqS)系统进行对比. 在 SeqS 中,每个任务都加载完整模型的副本,而 PetS 则通过处理轻量级的 PET 任务来高效支持多任务.

对于每个平台,我们通过如下方式测试其最大支持的任务数: 首先系统会加载 T 个任务(对于 PetS,每个任务属于一个随机的 PET 类型),然后测试系统是否能够处理 32 个随机生成的查询(每个查询长度为 128). 如果没有内存溢出(out-of-memory, OOM)问题,我们认为系统至少可以支持 T 个任务,可以不断增加 T 以测试极限.

表 3 展示了不同模型在不同平台上使用 SeqS 和 PetS 这 2 种框架进行推理支持的最大任务数. 与传统系统(这里我们以 TurboTransformers 作为代表性基线)相比, PetS 支持的并发任务数量增加了 4 倍(Jetson-TX2 上的 Bert-large)到 26 倍(Tesla-V100-32 GB 上的 DistillBert). 这主要是因为采用的共享权重的推理方式可以极大节省推理的显存开销. 因此,在将多个基于 Transformer 的应用部署到从边缘计算到云计算的各种场景时, PetS 可以大大节省硬件成本. 此外,即使调用了数百到数千个任务,它也避免了模型换入换出的低效率问题.

Table 3 Supported Task Numbers

表 3 支持的任务数

平台	DistillBert	Bert-base	Bert-large
	SeqS/PetS	SeqS/PetS	SeqS/PetS
Jetson-TX2	34/504	17/180	3/12
GTX-1080Ti-11GB	56/1 336	28/588	7/126
Tesla-V100-32 GB	170/4 344	85/2 164	25/560

3.2.2 吞吐量提升

PetS 通过统一的表示和专门的 PIE 实现了不同任务的统一批处理. 因此,我们评估了 PetS 在不同场景下的吞吐量,以每秒查询数(queries-per-second, QPS)表示. 我们在 Jetson-TX2, GTX-1080Ti-11GB, Tesla-V100-32GB 平台上加载了 4~32, 16~64, 32~128 个随机任务. 对于每个任务,我们生成具有 3 种固定形状的查询,具有相同形状的所有查询一起执行为一个批次,我们采用 TurboTransformers 运行单个任务作为基准. 如图 8 所示,在 GTX-1080Ti-11GB 和 Tesla-V100-32GB 平台上,与单任务服务基线相比, PetS 的

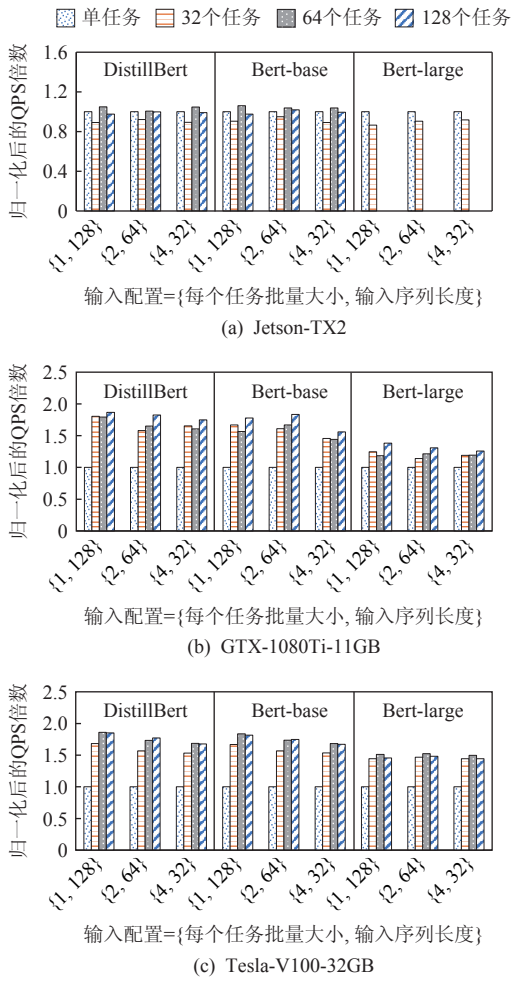


Fig. 8 Throughput improvement experiments on various platforms

图 8 多平台吞吐量提升实验

吞吐量提高了最高 1.87 倍 和 1.86 倍，平均提高了 1.53 倍 和 1.63 倍 . 我们注意到，在 Jetson-TX2 上，PetS 相比单任务服务未能获得明显的加速. 这是因为 Jetson-TX2 的计算资源有限（256 个 CUDA 核心），因此很难从批处理推理中获益. 类似地，在 GTX-1080 Ti 和 Tesla-V100 上，我们观察到 Bert-large 模型的加速比要低于 Bert-base/DistillBert. 这是因为 Bert-large 具有更大的尺寸（参见表 1），容易用满 GPU 的计算资源，导致降低批量推理的好处.

3.2.3 执行时间分析

为了进一步分析 PetS 在吞吐量上胜过基线系统的原因，我们对 PetS 和 SeqS 在 GTX-1080Ti-11GB 上的执行时间进行了细分. 我们设置了 2 个工作负载，并评估了包含 Bert-base 和 Bert-large 模型的 8 个随机任务. 如图 9 所示，通过共享算子的批处理执行，PetS 将非 PET 算子（包括注意力操作和共享线性层的计算）的速度提高了 2.17~3.28 倍. 由于采用了稀疏矩阵乘法库，PET 算子仅占总执行时间的 27.4%~41.3%. 因此，总体执行时间仍远远小于 SeqS.

3.2.4 内存占用分析

我们还对 PetS 和 SeqS 在 GTX-1080Ti-11GB 上的显存占用进行了分析，以揭示 PetS 具有出色可扩展性的原因. 以配置 {每个任务批量大小，输入序列长度} = {1, 128} 为例，在图 10 中绘制了模型权重和数据的 GPU 显存消耗情况. 我们可以看到，单个任务的权重参数约占用 0.35 GB 显存. 对于 SeqS，显存消耗随任务数量线性增长. 当任务数量达到 32 时，SeqS 的权重参数超过 11 GB，导致 GTX-1080Ti-11GB 出现 OOM 错误. 相反，对于 PetS，只有 PET 参数的显存占用随任务数量的增加而增加（通常不超过共享权重的 5%）. 因此，64 个任务的总显存占用量不超过 GPU 显存的 40%，证明 PetS 可以支持更多的任务. 请注意，16 个任务和 32 个任务的显存占用相同，但是 64 个任务的数据显存消耗比 32 个任务的多 3 倍. 这是因为我们使用 NVIDIA CUB 设备显存分配器进行动态数据显存管理，所分配的设备显存并不严格与数据大小成比例.

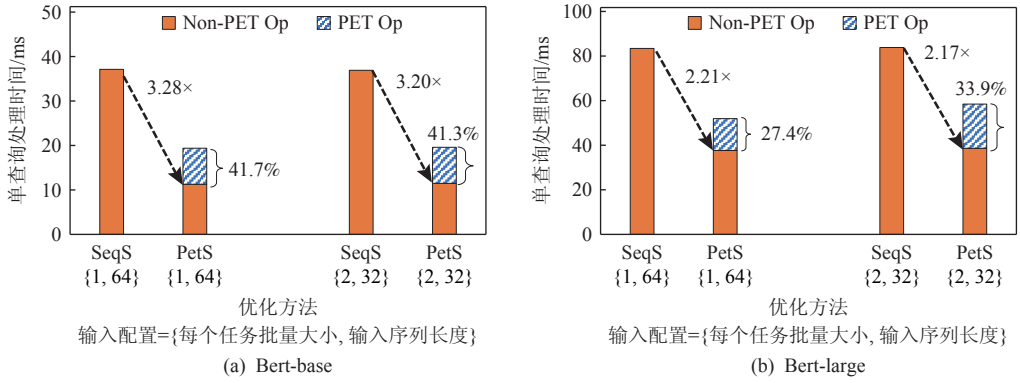


Fig. 9 Execution time-consuming breakdown

图 9 执行耗时分解

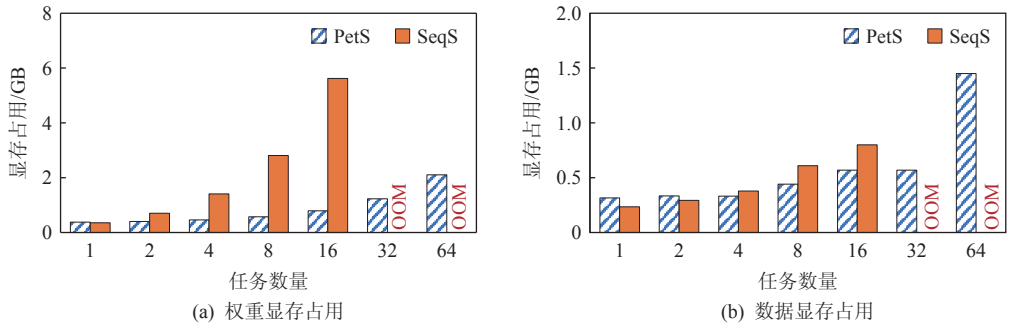


Fig. 10 GPU memory overhead comparison

图 10 GPU 显存开销对比

3.2.5 与并行推理系统的比较

除了 SeqS 之外, 本实验将 PetS 的吞吐量与传统的并行服务系统(parllel serving system, ParS)进行比较, 以展示其优越性. 为了实现 ParS, 本实验将每个任务特定的模型(Adapter, MaskBert, Diff-Pruning, BitFit 中的一种)放在唯一的 CUDA 流中, 以并行方式运行所有模型. 所有结果均在 GTX-1080Ti-11GB 平台上收集. 如图 11 所示, 本实验评估了多个请求配置, 用{每个任务批量大小, 输入序列长度}说明 PetS 的通用性. 本实验在 SeqS, ParS , PetS 上运行每个请求配置的不同任务数. 所有结果均相对于 SeqS 基线进行归一化. 当任务数量较小(1~4)时, PetS 无法胜过 ParS, 这是因为 PetS 需要额外的 PET 操作. 尽管 PetS 的共享权重部分也可以利用并行硬件, 但 PET 操作的开销无法通过有限的并行性来抵消. 随着任务数量的增加, PetS 的优势开始显现. 当任务数量达到 16 时, PetS 相对于 ParS 平均加速了 17.7%. 当任务数量继续增多时, 基线方法产生了显存溢出的错误, 而 PetS 则可以支持更多的任务并行执行.

3.3 单卡调度策略性能评估

在实际的多任务服务场景中, 输入查询通常具有可变的序列长度和 PET 类型. 简单地对这些查询进行批处理可能无法达到理想的吞吐量. 我们提出的协同批处理 (coordinated batching, CB) 策略, 旨在通过协调共享操作和 PET 操作来改善 PetS 在任意输入上的性能. 为了评估 CB 的效果, 我们对具有可变序列长度和 PET 类型的工作负载进行了测试, 并将 CB 与 3 种基线批处理策略进行了比较.

1)固定大小(Fixed)批处理. 将查询放入池中以形成固定大小的批次, 不考虑它们的 PET 类型和序列长度.

2)α-only 批处理. 仅使用 α 模型动态地对查询进行批处理. 这种策略类似于 TurboTransformers 的智能批处理.

3)β-only 批处理. 仅使用 β 模型动态地对查询进行批处理. 也就是说, 在图 6 中, 只会执行第 1 步, 获得的小批次将直接发送到 PIE 进行执行.

为了模拟实际情况, 我们假设查询的长度服从高斯分布. 为了不失一般性, 我们将长度均值设置为 32, 将长度标准差设置为 1~8, 并发任务数从 32 增加到 128. 每个任务被分配给一个随机的 PET 类型. 对于每个测试用例, 我们将 1 024 个查询放入查询池中. 对于池中的每个查询, 我们随机将其分配给一个已注册的任务.

如图 12 所示, 提出的 CB 策略相比固定大小批处理和 β-only 批处理速度平均提高 1.52 倍和 1.27 倍. 当标准差从 1 增加到 4 时, CB 相比 α-only 批处理也实现了高达 1.14 倍 (平均 1.06 倍) 的速度提升. 对于

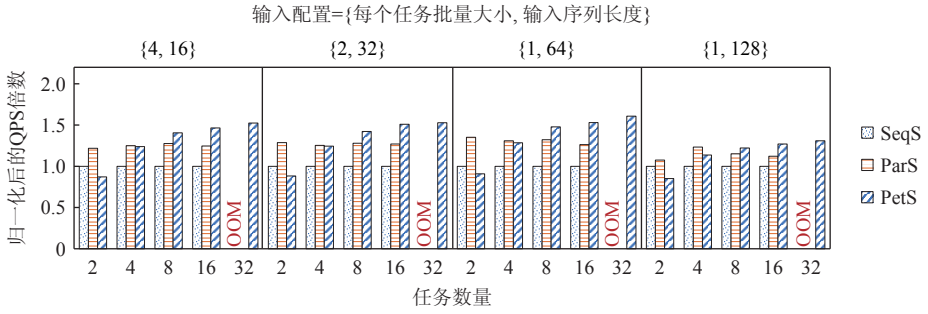


Fig. 11 Compared with ParS

图 11 与 ParS 的对比

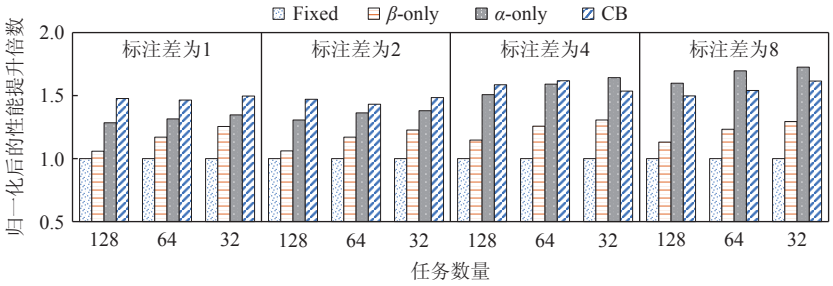


Fig. 12 Comparison of batching strategies

图 12 批处理策略对比

具有大方差的输入查询, CB 的吞吐量低于 α-only 批处理. 我们推断, 由 β 模型引导的第 1 阶段组批策略可能会将某些长度差异较大的查询放入同一个批次中, 在第 2 阶段中这种差异可能会被放大, 因为共享算子通常占据总执行时间的大部分. 相反, 如果输入查询具有中等或较小的方差, 则 2 个阶段批处理接近最优. 因此, 为了在任意输入上获得最高性能, 我们可以测量输入长度的方差来决定批处理策略.

3.4 基于动态迁移的多卡负载均衡策略评估

在多 GPU 场景下, 不同下游任务在一段时间内可能会存在被调用频次不同的问题, 从而引起不同 GPU 实例间的负载不均衡的问题. 在 2.3.2 节中本文结合 PET 算法的特性, 提出了基于 PET 热迁移的负载均衡策略. 为了评估该策略的效果, 我们在 1 台具有 8 卡 V100 的 DGX 服务器上部署了多个 PetS 实例并且通过 NVIDIA NCCL 库来在多卡间经过 NVLINK 链路传输需要调度的 PET 权重. 对于多台机器间的负载均衡, 我们可以使用 RDMA 或者 TCP 来传输 PET 权重, 但是由于实验硬件限制, 本文主要探讨单机内多卡的负载均衡问题.

3.4.1 固定迁移频率下的动态负载均衡性能

为了测量不同场景下 PET 热迁移带来的性能提升, 本文设置了 2 种配置, 即 4 GPU 实例和 8 GPU 实例, 并且加载了 128~4 096 个不同的 PET 任务. 在实际测试中, 我们将 1×10^6 个请求按照泊松分布分配到不同任务的任务队列中, 并且在运行过程中按照每 10 秒迁移 1 次的频率执行动态 PET 迁移, 并且测量整个系统在处理过程中的总吞吐量.

如图 13 所示, 通过 PET 任务热迁移, 系统的吞吐量在 4 卡和 8 卡的情况下提升了最多 13% 和 29%. 我们发现 8 卡比 4 卡提升得多, 这是因为当任务分配到更多卡上时, 其负载的不均衡性更加严重. 同理, 我们发现当任务变多时, 如 4 096 个任务, 通过 PET 迁移带来的性能提升显著变小, 这是因为当每个 GPU 中的任务变多之后, GPU 实例之间的数据分布差距会变小, 负载的不均衡性降低. 即便如此, PET 迁移仍然带来了 3%~4% 的性能提升.

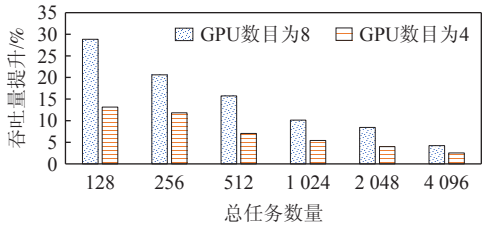


Fig. 13 Throughout improvement brought by PET migration

图 13 通过 PET 迁移带来的吞吐提升

同时, 我们也测量了 PET 热迁移带来的额外开销. 如图 14 所示, 我们测量了首轮 PET 迁移(首轮 PET 迁移的数据量最大, 稳定后的每轮迁移量较小)带来的开销. 随着总任务数量的增长, PET 的迁移总耗时增加, 但是由于 PET 本身参数量较少, 总的迁移时长低于 120 ms, 考虑迁移的时间间隔为 10 s, 其最大的开销占比为 1.2%. 并且当首轮迁移完毕后, 整个系统处于一个较均衡状态, 后面若干轮的迁移数据量约为首轮的 1/10, 其对性能的开销基本可以忽略.

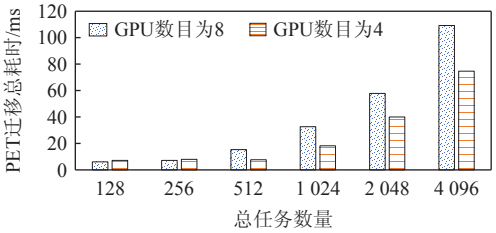


Fig. 14 Time overhead of PET migration

图 14 PET 迁移的时间开销

3.4.2 迁移频率动态调整算法性能测试

对于输入请求分布存在较大变化的场景, 2.3.2 节提出了一个算法来进行迁移频率的动态调整. 为了测试该算法的效果, 本实验设置了时长来控制请求分布的变化. 每当经过一定时长, 重新生成不同任

务的负载分布,以模拟分布产生较大变化的场景. 本实验的时长设置为 4~128 s.

对于 2.3.2 节中提出的算法,本实验将迁移间隔 $\phi_{\min}$ 和 $\phi_{\max}$ 分别设置为 0.2 s 和 20 s.最差不均衡阈值 θ 设置为 0.9, $\Delta\phi$ 设置为 0.5 s. 对于固定迁移间隔的方法,设置迁移间隔为 10 s. 系统的总任务数量设置为 128, GPU 数设置为 8.

如图 15 所示,对于不同时长,2 种策略的性能表现有显著的不同. 对于固定迁移间隔的方法,当输入分布的时长变化频率快于其迁移频率时,迁移非但没有性能提升,反而会略微降低系统的性能. 这是因为迁移间隔设置太大,其通过策略估计的负载均衡信息时效性不好. 对于动态迁移算法,由于可以根据迁移的效果动态调整迁移的间隔,其可以快速适应不同时长的场景. 对于时长较大的情况,如 128 时,2 种策略表现相当. 但是由于动态策略可以通过适当降低迁移频率来进一步降低迁移开销,其相对固定间隔方法,总吞吐量有一定的优势.

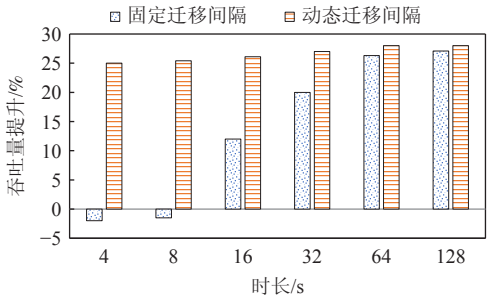


Fig. 15 Comparison of fixed migration threshold and dynamic migration threshold

图 15 固定迁移间隔与动态迁移间隔对比

4 结 论

本文提出了 PetS, 一个用于多任务 PET 推理服务的可扩展框架. PetS 提供了 PET 任务的灵活管理和高吞吐服务. 为了达到这个目的,我们首先提出了一个统一的算法表征,从而把不同的 PET 任务放到同一个框架中. 然后设计了一个 PET 推理引擎,通过批处理的方式计算不同的任务. 为了进一步提高系统吞吐量,我们提出了一种协调的批处理策略,同时考虑了输入的长度、PET 任务类型以及系统负载平衡. 为了提升多卡部署的负载均衡,我们创新性地提出了基于 PET 实时迁移的负载均衡机制. 我们在多个平台上评估了 PetS, 包括边缘端、桌面端和服务器端 GPU. 全面的实验证明, PetS 支持多达 26 倍的并发任务,并在桌面和服务器 GPU 节点上分别提高了 1.53 倍和 1.63 倍的服务吞吐量. 在多 GPU 场景下,我们的负载均衡策略可以提升多达 29% 的吞吐量.

作者贡献声明: 魏学超和周哲提出论文的思路和实现主要代码,以及部分论文撰写;徐盈辉、张洁靖、谢源、孙广宇参与论文的讨论和修改. 魏学超和周哲为共同第一作者.

参 考 文 献

[1] Devlin J, Chang M, Lee K, et al. BERT: Pre-training of deep bidirectional transformers for language understanding [C] //Conf of the 34th North American Chapter of the Association for Computational Linguistics. Stroudsburg, PA: ACL, 2019: 4171−4186

[2] Radford A, Wu J, Child R, et al. Language models are unsupervised multitask learners [EB/OL]. San Francisco, CA: OpenAI, 2019[2024-10-04]. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[3] Brown T B, Mann B, Ryder N, et al. Language models are few-shot learners [C] //Proc of the 34th Int Conf on Neural Information Processing Systems. New York: ACM, 2020: 1877−1901

[4] Radford A, Narasimhan K, Salimans T, et al. Improving language understanding by generative pre-training [EB/OL]. San Francisco, CA: OpenAI, 2018[2024-10-04]. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[5] Liu Yinhan, Ott M, Goyal N, et al. Roberta: A robustly optimized BERT pretraining approach [J]. arXiv preprint, arXiv: 1907.11692, 2019

[6] Yang Zhilin, Dai Zihang, Yang Yiming, et al. XLNet: Generalized autoregressive pretraining for language understanding [C] //Proc of the 33rd Int Conf on Neural Information Processing Systems. New York: ACM, 2019: 5753−5763

[7] Raffel C, Shazeer N, Roberts A, et al. Exploring the limits of transfer learning with a unified text-to-text transformer[J]. The Journal of Machine Learning Research, 2020, 21(1): 5485−5551

[8] Zhang Sunan, Roller S, Goyal N, et al. OPT: Open pre-trained transformer language models [J]. arXiv preprint, arXiv: 2205.01068, 2022

[9] Liu Ze, Lin Yutong, Cao Yue, et al. Swin Transformer: Hierarchical vision Transformer using shifted windows [C] //Proc of the 35th IEEE/CVF Int Conf on Computer Vision (ICCV). Piscataway, NJ: IEEE, 2021: 10012−10022

[10] Dosovitskiy A, Beyer L, Kolesnikov A, et al. An image is worth 16x16 words: Transformers for image recognition at scale [J]. arXiv preprint, arXiv: 2010.11929, 2020

[11] Hu Qinghao, Ye Zhisheng, Wang Zerui, et al. Characterization of large language model development in the datacenter [C] //Proc of the 21st USENIX Symp on Networked Systems Design and

Implementation. Berkeley, CA: USENIX Association, 2024: 709−729

[12] Fang Jiarui, Yu Yang, Zhao Chengduo, et al. TurboTransformers: An efficient GPU serving system for Transformer models [C] //Proc of the 26th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2021: 389−402

[13] Crankshaw D, Wang Xin, Zhou Guilio, et al. Clipper: A low-latency online prediction serving system [C] //Proc of the 14th USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2017: 613−627

[14] Gao Pin, Yu Lingfan, Wu Yongwei, et al. Low latency RNN inference with cellular batching [C/OL] //Proc of the 13th EuroSys Conf. New York: ACM, 2018[2025-01-18]. https://dl.acm.org/doi/10.1145/3190508.3190541

[15] Guo Demi, Rush A, Kim Y. Parameter-efficient transfer learning with diff pruning [C] //Proc of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th Int Joint Conf on Natural Language Processing. Stroudsburg, PA: ACL, 2021: 4884−4896

[16] Houlsby N, Giurgiu A, Jastrzebski S, et al. Parameter-efficient transfer learning for NLP [C] //Proc of the 36th Int Conf on Machine Learning. New York: PMLR, 2019: 2790−2799

[17] Zaken E B, Ravfogel S, Goldberg Y. BitFit: Simple parameter-efficient fine-tuning for Transformer-based masked language-models [C/OL] //Proc of the 60th Annual Meeting of the Association for Computational Linguistics. Stroudsburg, PA: ACL, 2022[2025-01-18]. https://aclanthology.org/2022.acl-short.1.pdf

[18] Zhao Mengjie, Lin Tao, Mi Fei, et al. Masking as an efficient alternative to finetuning for pretrained language models [C] //Proc of the 58th Annual Meeting of the Association for Computational Linguistics. Stroudsburg, PA: ACL, 2020: 2226−2241

[19] Hu E, Shen Yelong, Wallis P, et al. LoRA: Low-rank adaptation of large language models [J]. arXiv preprint, arXiv: 2106.09685, 2021

[20] NVIDIA. Fast Transformer [EB/OL]. 2021[2024-10-04]. https://github.com/NVIDIA/FasterTransformer

[21] Wang Xiaohui, Xiong Ying, Wei Yang, et al. LightSeq: A high performance inference library for transformers [J]. arXiv preprint, arXiv: 2010.13887, 2020

[22] Gururajan A K, Lopez-Cuena E, Bayarri-Planas J, et al. Aloe: A family of fine-tuned open healthcare LLMs [J]. arXiv preprint, arXiv: 2405.01886, 2024

[23] Gupta A, Shirgaonkar A, Balaguer A D L, et al. RAG vs fine-tuning: Pipelines, tradeoffs, and a case study on agriculture [J]. arXiv preprint, arXiv: 2401.08406, 2024

[24] Yang Hongyang, Liu X, Wang C D. FinGPT: Open-source financial large language models [J]. arXiv preprint, arXiv: 2306.06031, 2023

[25] Romero F, Li Qian, Yadwadkar N J, et al. INFaaS: Automated model-less inference serving [C] //Proc of the 29th USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2021: 397−411

[26] Shen Haichen, Chen Lequn, Jin Yuchen, et al. Nexus: A GPU cluster engine for accelerating DNN-based video analysis[C] //Proc of the 27th ACM Symp on Operating Systems Principles. New York: ACM, 2019: 322−337

[27] Sidhu S, Wing J, Japi A. Rafiqi: A GPU-based deep learning model serving system [R]. Berkeley, CA: University of California, 2020

[28] NVIDIA. Triton inference server [EB/OL]. 2018[2024-10-04]. https://developer.nvidia.com/nvidia-triton-inference-server

[29] Google. TensorFlow serving [EB/OL]. 2016[2024-10-04]. https://github.com/tensorflow/serving

[30] Prasanna S, Rogers A, Rumshisky A. When BERT plays the lottery, all tickets are winning [J]. arXiv preprint, arXiv: 2005.00561, 2020

[31] Mao Yuning, Mathias L, Hou Rui, et al. UniPELT: A unified framework for parameter-efficient language model tuning [C] //Proc of the 60th Annual Meeting of the Association for Computational Linguistics. Stroudsburg, PA: ACL, 2022: 6253−6264

[32] Gale T, Zaharia M, Young C, et al. Sparse GPU kernels for deep learning [C/OL] //Proc of the 33rd Int Conf for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2020[2025-01-18]. https://dl.acm.org/doi/10.5555/3433701.3433723



Wei Xuechao, born in 1988. PhD. His main research interests include computer architecture and electronic design automation.

魏学超, 1988 年生. 博士. 主要研究方向为计算机体系结构、电子设计自动化.



Zhou Zhe, born in 1997. PhD. His main research interests include computer architecture, near data processing, domain specific accelerator, machine learning system and interconnection techniques.

周　哲, 1997 年生. 博士. 主要研究方向为计算机体系结构、近数据处理、领域专用加速器、机器学习系统和互联技术.



Xu Yinghui, born in 1973. PhD. professor, PhD supervisor. His main research interests include generative AI, logic reasoning, and large language model for life science.

徐盈辉, 1973 年生. 博士, 研究员, 博士生导师. 主要研究方向为生成式 AI、逻辑推理、生命科学大语言模型.



Zhang Jiejing, born in 1986. Bachelor. Senior Member of Technical Staff. Director of Alibaba Tongyi Lab heterogeneous computing team. His main research interest includes deep learning inference.

张洁靖, 1986 年生. 学士. 高级技术专家. 阿里巴巴通义实验室异构智能技术负责人. 主要研究方向为深度学习推理.



Xie Yuan, born in 1974. PhD. IEEE fellow, ACM fellow, AAAS fellow, Chair professor. His main research interests include computer architecture, VLSI, electronic design automation, machine learning.

谢　源, 1974年生. 博士, IEEE, ACM, AAAS会士, 讲席教授. 主要研究方向为计算机体系结构、超大规模集成电路、电子设计自动化和机器学习.



Sun Guangyu, born in 1981. PhD, professor, PhD supervisor. His main research interests include electronic design automation for domain specific architecture and new devices, architecture/circuit/device codesign and automation .

孙广宇, 1981年生. 博士, 教授, 博士生导师. 主要研究方向为面向领域定制架构和新器件电子设计自动化、架构/电路/器件协同设计与自动化.